**Functional Programming and Lambda**

Functional programming is a paradigm change.

A functional interface is an interface with only one abstract method all other method will be either default or static. We can annotate the interface with @FunctionalInterface, but it’s a marker annotation only.

That single method is called function descriptor.

Rather creating Anonymous class implementations, we can use lambda expression if the interface is a functional interface.

We can also create our own functional interface, but for simplicity java developers had already created many functional interfaces.

Some of the useful functional interfaces are

Runnable

Predicate<T>

BiPredicate<T,R>

Function<T,R>

BiFunction<T,R,S>

Consumer<T>

BiConsumer<T,R>

Supplier<T>

UnaryOperator<T>

BinaryOperator<T>

There are some other functional interfaces to work with the primitive types like IntBinaryOperator, IntConsumer, IntFunction, IntPredicate, IntSupplier, IntToDoubleFunction, IntToLongFunction. As the upper functional interfaces works with Object types mainly, so to use stream of Integers there will be autoboxing and unboxing which is in efficient, that’s why these primitive functional interfaces are also used.

We can directly instantiate a functional interface with lambda expression like the following

**Predicate<Integer> predicate = num -> num % 2 == 0;**

**Consumer<Integer> consumer = System.out::println;**

**num -> num % 2 == 0** is a lambda expression, with which we can direct add the code here.

If there is only one parameter, then we don’t need parentheses. If there one statement, then we don’t need to add it in curly braces. We don’t need to specify the type of the input it will be inferred automatically.

**System.out::println** is a method reference.

We can use method reference for static methods, member methods or constructor. Just that the signature should match.

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

numbers

.stream()

.filter(num -> num % 2 == 0)

.map(num -> num \* 2)

.forEach(System.***out***::println);

.stream() method makes a iterable to stream object

.filter() method takes predicate functional interface to filter out the object. It will take an object and return Boolean.

.map() method takes Function functional interface to transform the object to another object. It will take an object and return another or same object.

.forEach() method takes Consumer functional interface to consume an object. It is void return type. It will only take an object. It is a terminal operation.

.reduce is a reduction operation which takes an identity or a stating element and a BiFunction functional interface which takes two inuts and returns one output. We can use multiple cores and thread to perform reduce if it’s a big set of data.

Integer sum = numbers

.stream()

.reduce(0, (a, b) -> a + b);

0, (a, b) here 0 is the identity or the starting point, a is the aggregate or the output of the previous operation and b is the current item. If we are only doing sum then we can use Integer:: sum or the sum() method.

.distinct() method removes all the duplicate elements.

.sorted() method makes the stream sorted. By default, it will sort in increasing manner. This is an overloaded method which takes Comparator functional interface. We can also assign our custom sorting logic with this. We can also assign Comparator.naturalOrder() or Comparator.reverseOrder().

We can also assign our own logic using Comparator.comparing like sorted

(Comparator.comparing( str -> str.length() ) )

numbers

.stream()

.distinct()

.sorted(Comparator.*naturalOrder*())

//.sorted(Comparator.reverseOrder())

//sorted()

.forEach(System.***out***::println);;

.collect() is a terminal operation where we can collect the stream of data and make it as list or set or map anything.

List<Double> squareRoots =numbers

.stream()

.map(Math::sqrt)

.collect(Collectors.toList());

Intermediate operation returns a Stream. Terminal operation returns anything other than Stream.